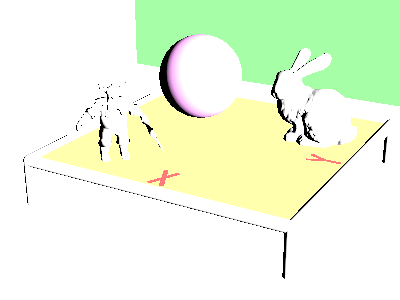
**Project Report**

After finishing the project, I used a few metrics to verify all collision data was accurate. I made a very approximate lighting algorithm to see if it looked very roughly like a rendered scene should. The equation that I found useful was multiplying the Kd with the dot product of the Normal and the center of the light sphere and then dividing the whole thing by 2 (0.5\*dot(N,Lapprox) \* Kd). The division by 2 is important as without it the whole image looks overexposed and is hard to judge the correctness. Figure 1 shows the equation without the division by 2. Comparing this to figure 2, which has the division, it can be seen how the division helps. The second figure has smaller highlights throughout, making more detail visible. The first figure also has parts where the Kd value is completely overwritten with white, like with the blue wall.

The second figure shows the results of the approximate lighting calculation. It shows the arc of shadow on the left side of the sphere and the line of shadow on the cylinder. It also shows the more complex shadows of the triangle meshes. This was my default method for testing correctness throughout the debugging process, only moving onto more specific output data when needed.

A picture containing text, table, worktable

Description automatically generated

Figure : Adjusted Lighting Approximation Figure : Initial Lighting Approximation

The other metrics I used was outputting the Kd, ray intersection t value(as (t-5)/4), and the absolute value of the normal, in figures 3, 4, and 5 respectively. I found outputting the Kd value to be the most useful view for initial debugging. For each shape I first implemented the intersection algorithm without adding the normal calculations. My first step was always to make sure that the rays were intersecting the object on all the pixels that it was supposed to. Only after the flat Kd output was working did I start implementing the normal for the shape. As for the other two specialized outputs, I mostly used them only for correctness verification after each shape algorithm was implemented and looked finished in the lighting equation output. I found that whenever the approximate lighting calculation output looked right the t value and normal value outputs also were correct so these views weren’t as useful but important for verifying the results.

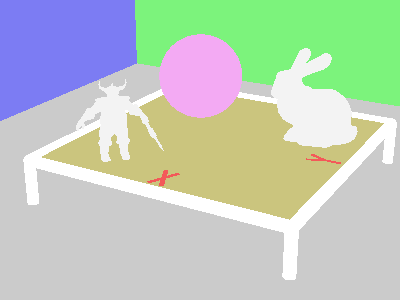
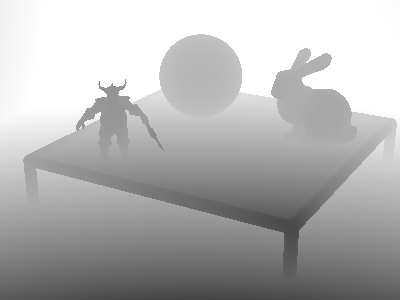


Figure : Kd values

A picture containing text, vector graphics, light

Description automatically generated

Figure 4: t values Figure 5: N values

After implementing the spatial data structure there was a very noticeable improvement of execution speed. For testing the speed of execution the full scene was rendered, including the Stanford bunny, to make the difference noticeable. Without the spatial data structure, the image took anywhere from 180 to 200 seconds, or a little over 3 minutes, to execute. After implementing the spatial structure, the execution time went down to around 53 seconds, consistently less than a minute every time. Despite the speedup running the code with and without the spatial data structure still resulted in the output seen in figure 2. This makes since as the structure should change anything about the data being returned, which confirms that the structure was implemented correctly.

One extra feature that I added to the codebase was integrating the real-time rendering into the raytracing code instead of excising it. The way I did this was that instead of the program immediately doing the raytracing algorithm on start, it loads the real-time simulation instead. After exiting the program with escape, the raytracing algorithm initiates and outputs its findings. By pressing escape immediately, the raytracer will use the default camera position, which is the same one found in all the sample outputs from the handouts. This default view is useful for comparing with expected output however it has its limitations.

The real-time simulation, on exit, passes the camera information to the raytracer. This allowed me to move the camera in smoothly in the real-time application and then very easily have the raytracer use the new camera position. The only other way to achieve this effect would be by switching out hardcoded camera positions, which would need to be found through guesswork and many recompilations in a very tedious process. One example of the usefulness of this feature can be seen in figure 6, which shows the back of the scene from behind. This angle make it easy to confirm that the dark side of the objects are also being rendered correctly, even though they cannot be seen from the default view.

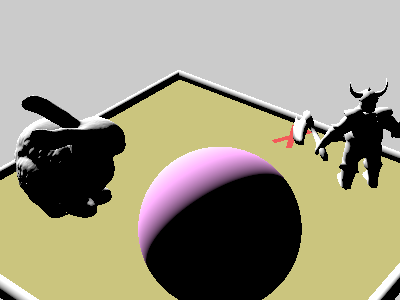


Figure 6: Scene From Behind

The most valuable part of the real-time feature, however, is in its debugging capabilities. For instance, the bug the gave me the most trouble during implementation involved the triangle meshes not rendering fully. From the default angle it looked as though the dwarf’s face was black, which could have been an indication that the normal values of some of the triangles were incorrect. However, figure 7 shows the dwarf from a side angle which make it clear that the face triangles are not being rendered at all, narrowing down the scope of where the bug could be. Following professor Herron’s advice, I then subdivided the mesh until I was down to a single triangle that was not being rendered. I was able to used the adjustable camera to move the camera up close to the single triangle and render from there. The triangle didn’t show up in the raytraced image but because it was so big in the real-time camera view, this confirmed that it was not being rendered and made it easy to step through the code and determine where it was returning a false negative in the intersection algorithm. After finding and fixing the issue, I was then able to use the adjustable camera to render the same view and confirm that the triangle was now being rendered correctly (as seen in figure 8) before returning to the full dwarf mesh. I could then retry the same angle as figure 7 to confirm the whole face was rendering correctly.

 Icon

Description automatically generated with medium confidence

Figure 7: Broken Dwarf Side View Figure 8: Fixed Triangle Closeup



Figure 9: Fixed Dwarf Side View

The adjustable camera also underlines the usefulness of the spatial data structure as figure 9 without the spatial structure took about 3 minutes to render since the raytracing algorithm has to brute force test against every shape, taking the same amount of time no matter the camera angle. With the spatial data structure figure 9 only took 5 seconds to render as it was able to skip over the Stanford bunny that was behind the camera.